Data Structures

Assignment 1

Linked Lists, Array Lists

Due: 11pm on Sunday, Feb 19, 2017

This assignment has two parts and each part comprises of two tasks. In the first part, you will implement: (i) a linked list and (ii) an array-based list. In the second part, you will implement: (i) a stack and a queue ADT using your linked list implementation and (ii) solve some problems using your implementation of a stack and a queue.

You may test your implemented data structures with the test cases that we have provided you (test1.cpp, test2.cpp, test3.cpp, test_isPalindrome.cpp and test_min.cpp). **Please note that your code must compile at the mars server under the Linux environment.**

Start early as the assignment would take time.

The course policy about plagiarism is as follows:

1. Students must not share their program code with other students.
2. Students must be prepared to explain any program code they submit.
3. Students must indicate any assistance they received.
4. Students cannot copy code from the Internet
5. All submissions are subject to automated plagiarism detection.

Students are strongly advised that any act of plagiarism will be reported to the Disciplinary Committee

Note: *In the following write-up, anything in green either refers to something in the actual code or a file name.*

PART 1

TASK 1 (Linked List):

In this part of the assignment, you are required to implement a linked list class that contains various member functions. The basic layout of this linked list class is provided to you in *LinkedList.h.*

The Template ListItem in the LinkedList.h file represents a node in the linked list. The class LinkedList (see LinkedList.h) implements the linked list, which contains a pointer to the head of the linked list and function declarations. Test cases for the LinkedList class are provided in test1.cpp, make sure you pass all of them before submission.

NOTE: *When implementing the member functions, make sure you handle all error conditions such as deletion from an empty linked list*.

Member functions:
This section defines the purpose of the member functions in the List class for which you must write the code

List()

This is simply the default constructor of the List class

List(const List<T>& otherList)

This is the copy constructor of the List class which when provided with a pointer to another list otherList constructs a linked list with the same elements as otherList.

~List()

Destructor for the List class. Deletes all the nodes in the list and frees the memory allocated to the linked list

void insertAtHead(T item)

Function which inserts an item of type T (remember the basic unit is a template) at the head of the list.

void insertAtTail(T item)

This function inserts an item of type T at the tail of the list.

void insertAfter(T toInsert, T afterWhat)

This function first goes through the linked list to find the item afterWhat. When this is found it then inserts toInsert after the afterWhat linked list node.

void insertSorted(T item)

This function adds item in its correct position in a linked list which is already sorted in ascending order (i.e., the head would have smallest value and tail would have the largest value). Note that after the insertion, the resulting list should remain sorted.

ListItem<T> *getHead()

Returns the head pointer of the linked list, returns NULL if empty.

ListItem<T> *getTail()

Returns the tail pointer of the linked list, returns NULL if empty.

ListItem<T> *searchFor(T item)

Returns the pointer to node contain the element item. The function returns NULL if the list is empty or the item is not found.

void deleteElement(T item)

This function deletes the first node containing the element item.

void deleteHead()

Deletes the first node (i.e., the head) of the linked list.

void deleteTail()

Deletes the tail node of the linked list.

int length()

This function returns the length of the linked list i.e., the number of nodes currently in the linked list.

void reverse()

This function reverses the order of the linked list (NODES must be swapped, NOT VALUES):
e.g., 1, 2, 3, 4, 5 ------> 5, 4, 3, 2, 1

void parityArrangement()

This function changes the order of nodes in the linked list in the following way:
e.g., 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ------> 1, 3, 5, 7, 9, 2, 4, 6, 8, 10

i.e., all the odd indexed nodes will be together at the front of the list, followed by all the even indexed nodes.

TASK 2 (ArrayList):

For this part of the assignment, you need to implement an ArrayList class that contains various member functions. The basic layout of this class is also provided to you in *ArrayList.h.*

You will need to provide similar functionality for the ArrayList class as the Linked List class above. However, note that the ArrayList uses a dynamic array to store data, therefore you must be careful to prevent overwriting or fragmentation (Empty spaces in between elements in the array) after insertions and deletions. Most functions for this part will have similar functionality to the

linked list class. You will need to double the array size when the size becomes equal to the array capacity. DO NOT change the initial size of the array. Test cases for ArrayList class are provided in test2.cpp.

PART 2

TASK 3 (Stacks and Queues):

For this task, you will use your linked list class to implement the stack and queue data structures.

Stack:

The Stack class contains a List type object. This means that all the member functions defined in the Linked List class would be accessible. (The same is the case for the Queue class)

Member functions:

This section defines member functions of the Stack class for which you have to write the code given in the header file stack.h.

Stack()

This is the default constructor of the Stack class.

Stack(const Stack<T>& otherStack)

This is the copy constructor for the Stack class which when provided with a pointer to another Stack, constructs a new Stack with the same elements as the existing otherStack object.

~Stack()

Destructor for the Stack class. Be sure to free any allocated memory.

void push(T item)

Function which inserts elements in the Stack object (which follows the Last-In First-Out (LIFO) policy for insertions and deletions).

T top()

Function which returns the value on top of the Stack i.e., the element that was last inserted. This function only returns the element without removing it from the Stack.

T pop()

Function which returns the value on top of the Stack (the element last inserted into the Stack) and in addition, removes the element from the Stack.

int length()

This function returns the number of elements in the Stack.

bool isEmpty()

This function returns true if the stack is empty and returns false otherwise.

Queue:

The Queue also contains a List type object, just like the Stack.

Member functions:

Queue()

This is the default constructor of the Queue class.

Queue(const Queue<T>& otherQueue)

Copy Constructor for the Queue class which when provided with a pointer to another Queue contructs a new Queue object with the same elements as the existing otherQueue object.

~Queue()

Destructor for the Queue class. Be sure to free any allocated memory.

void enqueue(T item)

Function which is used to add elements to the Queue object (which follows the First-In First-Out (FIFO) policy for insertions and deletions).

T front()

Function which returns the element at the front of the Queue i,e., the element in the queue which was inserted first. This function only returns the element without removing it from the Queue.

T dequeue()

Function which returns and removes the element at the front of the Queue.

int length()

This function returns the number of elements in the Queue.

bool isEmpty()

This function returns true if the queue is empty and false otherwise.

PART 3 (Applications):

TASK 4

In this task, you are required to implement a function called IsPalindrome(). Given a singly linked list, you are required to figure out if it is a palindrome or not. This function should return 1 if a linked list is a **palindrome**, otherwise return 0.

A palindrome is a word or phrase (in our case a list of numbers) that reads the same, backwards and forwards. For example:

5->8->2->7->2->8->5

NOTE: *Your solution must be in O(n) time.*

TASK 5

You have to design a stack which, in addition to push and pop, also has a function getMin() which returns the **minimum** of that element and all the items below that item in the stack.

You can change the implementation of your stack as much you like. This question has been designed to allow you to be as creative as possible with your implementation.

You need to pass the test cases provided in test_min.cpp. In the case of wrong implementation, you are most likely to encounter segmentation faults, so be prepared and have fun debugging.

NOTE: Push, pop and getMin should all operate in O(1) time.